NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

N175

DESIGN AND IMPLEMENTATION
OF A PROGRAM FAMILY FOR TYPE EVALUATION

by

Timothy B. Nachtsheim

December 1987

Thesis Advisor: Gordon H. Bradley

Approved for public release; distribution is unlimilted.

T239216

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
DESIGN AND IMPLEMENTATION OF A PROGRAM FAMILY FOR TYPE EVALUATION (u)

12 PERSONAL AUTHOR(S)
Nachtsheim, Timothy B.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1987 December	52

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Program family; type evaluation; reuseable software; information hidig; modularization; program environment

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

One approach to designing reuseable software is to consider a program to be a member of a family of programs that are related through function, purpose, or lineage. Numerical evaluation of expressions is a function that links many programming environments together, such as programming languages, operations research models, and spreadsheet applications. In parallel, we may have type evaluation of expressions in these environments, a function unually performed by hand. By considering the need for type evaluation in these environments to constitute a problem domain from which a program family can be generated, a design for a type evaluator to support the family members is developed. Several examples of environment specific implementations are provided, and the degree of reuseability through the approach is discussed.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. Gordon H. Bradley	22b TELEPHONE (Include Area Code) (408) 646-2359 22c OFFICE SYMBOL Code 52Bz

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

U.S. Government Printing Office: 1986—606-24

Approved for public release; distribution is unlimited.

Design and Implementation of a Program Family for Type Evaluation

by

Timothy B. Nachtsheim
Lieutenant, United States Navy
B.S., University of Washington, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1987

ABSTRACT

One approach to designing reuseable software is to consider a program to be a member of a family of programs that are related through function, purpose, or lineage. Numerical evaluation of expressions is a function that links many programming environments together, such as programming languages, operations research models, and spreadsheet applications. In parallel, we may have *type* evaluation of expressions in these environments, a function usually performed by hand. By considering the need for type evaluation in these environments to constitute a problem domain from which a program family can be generated, a design for a type evaluator to support the family members is developed. Several examples of environment specific implementations are provided, and the degree of reuseability attained through the approach is discussed.

Thesis
N175
c.1

TABLE OF CONTENTS

I. INTRODUCTION 6
A. MOTIVATION 7
B. IMPLEMENTATION 9
II. EXPRESSIONS AS A PROBLEM DOMAIN 10
A. ABSTRACTION 10
1. Defining the Problem Domain 10
2. Language of Expressions 11
B. TYPING IN EXPRESSIONS 12
1. Meaning of Type 12
2. The Process of Type Evaluation 13
III. PROGRAM DESIGN 15
A. DESIGNING FOR CHANGE 15
1. Design Principles 15
2. Identifying the Changes 15
B. PROGRAM DESIGN 17
1. Creating the Environment Through Declarations 17
Interpreting the Language 19
Lexical Analysis 19
Syntactic Analysis 22
2. Type Evaluation Phase 26
A Generalized Expression Grammar 26
Symbolic Type Evaluation and Conversion 29
IV. EXAMPLE FAMILY MEMBERS 34
A. REAL/INTEGER/BOOLEAN AS A PROTOTYPE 34
B. INTERVAL ARITHMETIC 39
1. Intervals as a Type 39
2. Implementing Intervals 41
C. THE DIMENSIONAL MODEL TYPE 43
1. Description of Operations Research Model 43
2. Model Example 46
V. CONCLUSION 49
LIST OF REFERENCES 51
INITIAL DISTRIBUTION LIST 52

LIST OF FIGURES

Figure 2-1. Type conversion inserted into tree 14
Figure 3-1. User Model 18
Figure 3-2. Elements of the Program Family 27
Figure 3-3. Program Family Core for Type Evaluator 33
Figure 4-1. Expression as Postfix Tree 38
Figure 4-2. Postfix Tree with Conversions Inserted 38
Figure 4-3. Interval Expression as Postfix Tree 42
Figure 4-4. Interval Postfix Tree After Evaluation 43
Figure 4-5. PostFix Expression Before and After Conversions 48

I. INTRODUCTION

Expressions occur in programming languages, operations research models, spreadsheet applications, and many scientific computer based systems. Expression evaluation is common to all of these, and the process from recognition of an expression to a determination of its value is a well understood one. In parallel however, we may have *symbolic type evaluation* , which is a determination of the *type* of an expression, an attribute that occurs in a pair with its value. Built-in value evaluation of expressions is common to many current systems, yet type evaluation of these expressions is mostly performed by hand. It is usually the task of the person forming an expression to ensure that it is well formed in the sense of type consistency between operators and operands prior to using it in these systems. Many systems and applications would benefit greatly from the inclusion of type evaluation, leading to more powerful and secure systems.

For example, many current operations research modeling systems are used infrequently and those intimate with the specifics of the system are no longer around when the model is used. Those who desire to use the model must learn or in some cases, re-learn, the details of the model in order to use it. By having a system type evaluator to perform data and expression validation, we can reduce the amount of time it takes to effectively use the system. If some part of the model is changed, for example, the typing of the data; a type evaluator would be able to validate the compatability of the modified data with the original model. Another example concerns executable modeling languages, which specify a type calculus for the types supported by the language. The calculus

suggested by Bradley and Clemence [1] supports a hierarchy of dimensional types that allows upward inheritance and conversion of types based on the outcome of applying operators in the system. By providing such a system with a type evaluator, we can check the composition of constraints and functions for consistency, as well as provide automatic type conversion.

In many modeling systems, the model and the data are in separate files. The separation of model and data is natural in systems where the model is developed and validated, and then many different problems are formed by combining the model with different data files. While this design allows the inclusion of the specification of sets and the value of coefficients with the data, there exists the danger that a particular data file may not conform to the model's type requirements [1]. By providing a type evaluator that is capable of performing conversions, a data file may be converted during input or output to conform to the specific type constraints of the model.

A. MOTIVATION

Parnas defines a *program family* as a set of programs whose common properties are so extensive that it is advantageous to study the commonalities between them before analyzing (or developing) individual members of the set [2]. From our research, type evaluation is a function that is common to many systems. Each of these systems has specific typing requirements and data structure definitions, yet the abstraction of the problem domain leads us to believe that type evaluation is not dependent on these issues.

If we say that the *process* of type evaluation does not change from one problem in the domain to another, then we may develop a family of programs with a common ancestor that supports the varied type and data instances. Such an approach has many benefits:

1. Since variations in applications are to be expected, such an approach will minimize the amount of development needed to produce programs for each application.

2. By making the process the key design problem, we may increase the portability of the design from one language to another

3. Once an ancestor program is developed, it can be used in a library of programs, available to application designers whose problem specifications fall within the domain

4. By considering all similar problems, we can better determine the basic design to solve the individual ones

Since the requirement for performing type evaluation exists in many systems, we would like to avoid developing multiple programs to handle the various different types that would be found in these systems. In today's environment of rapidly increasing software development costs, the approach of designing a family of programs that support many members from the same problem domain is of particular interest. Another critical issue concerns the movement within the Department of Defense to require the ADA language to be the primary programming language of all new systems. To fully utilize the basic premise of ADA, that is, modularization, we may design a program in such a way that many of the modules may be reuseable from one family member to another without modifications.

B. IMPLEMENTATION

In support of this idea, a program family for type evaluation is developed and implemented in a Pascal-like language. The initial design discussion centers on the core routines of this program family which can be used by every instance of the family. The second part of the discussion gives several example members from the family including one that is used in conjunction with an executable modeling language for a transportation model. These examples are fully running programs that take the core routines and add only the type-specific routines necessary to make a complete program.

II. EXPRESSIONS AS A PROBLEM DOMAIN

A. ABSTRACTION

1. Defining the Problem Domain

Abstraction is an intellectual tool that allows reuseability, extensibility, and generality to be incorporated into a design. Here we use it as a basis for program development. We first define the problem domain as the set of all computable expressions. These expressions may occur in many environments, such as mathematics, programming languages (e.g. PASCAL, FORTRAN, ADA), in operations research modeling systems, spreadsheets, etc. We can then further identify the problem domain as those environments which contain expressions and require the evaluation of them. By examining representative members of this set of problems, we observe that they may be viewed in abstraction to have certain common features. These are:

- there are processable elements of the domain
- there are process operations in the domain
- there exist relationships among the processable elements and the operations
- there are precedence relationships amongst the operations

Next, we analyze these features to determine which are the non- changing base objects of any expression. We call every processable element an *operand* and every process operation an *operator*. For the purpose of our discussion we restrict ourselves to unary and binary operators, as virtually all systems support these.

An operator in a language can be characterized informally as a token which specifies an operation of function evaluation. An operand is a token which specifies a value, either literally (as a constant) or indirectly (as an identifier). An expression is then a sequence of operands, operators, and grouping indicators (delimiters, such as parathesises) [3].

2. Language of Expressions

Every expression in the problem domain is composed of operands and operators. Since these operators and operands can be composed together to create expressions, we must have rules to do so. Typical rules as presented in Aho & Ullman [4] are:

1. A single operand is an expression.
2. If @ is a binary infix operator and E1 and E2 are expressions, then E1 @ E2 is an expression.
3. If @ is a unary prefix operator and E is an expression, then @E is an expression.

These rules then define the relationship among operators and operands. When we combine operators to form expressions, the order in which the operators are to be applied may not be obvious. For example, a + b + c can be interpreted left to right (left-associative) or from right to left (right-associative) [4]. We may also force the associativity of an expression through the use of parathesises, as is done in most programming languages. We can then add another rule,

4. If E is an expression, (E) is an expression.

Another relationship among operators is the notion of precedence. For example, the expression a + b * c can be grouped according to a chosen associativity rule, however the

associativities of + and * do not tell us which is preferred. We therefore use precedence rules to determine the proper grouping of operands and operators [4]. We then let these elements be the basis for all generalized expressions, and consider the rules for expression construction a formal grammar for a language.

B. TYPING IN EXPRESSIONS

1. Meaning of Type

The value evaluation of expressions is a well known subject, but our interest is in symbolic type evaluation of these expressions. We must answer, what is the meaning of the type of an expression, and what is involved in the process of type evaluation?

With each expression is associated a value , which is determined by the values of its operands, the functions satisfied by the operators, and the sequence in which the latter are applied to the values of their arguments [3]. If we consider instead of values of operands the *types* of the operands, then we may define the *type* of an expression to be the result of applying type sensitive operators to the operands of the expression. That is, when type evaluation is performed, the type of each operand is substituted for its symbolic name in applying operators, and then the expression is evaluated according to the semantics of the operators [5]. For example, in most programming languages, all operands have a type attribute, be it the usual well known ones of real, integer, boolean, character (base types), or some complex type, such as arrays or records whose fields are base types. These type attibutes of the language objects are then used in syntactic and semantic processing of language statements, including expression evaluation.

2. The Process of Type Evaluation

During compile time checking of an expression such as

a (of type integer) + b (of type real)

operand "a" is implicitly converted to a real type, and then during run-time the "+" operator is applied. What is in effect occuring during the compile time check is *type evaluation* , and this is the process we wish to identify.

To begin with, we note that the simple example expression is composed of two operands and a binary operator. During syntactic validation of any statement in any language, the completeness of the statement is checked, and we assume in this case that the binary operator has associated with it two operands and it is formed in the correct infix order. The next step is two-fold:

(1) Determine whether each operand is type correct for the operator
(2) Or, if not, whether the operand can be converted to a type consistent with that operator.

In step (1), various mechanisms are used in compilers to perform this function, for example, look-up tables that use the operator and the operands as entries into a matrix of allowed combinations, and returns a true-false answer. Since step (2) is really part of the answer to step (1), we can include an identification of which operand(s), if any, can or need to be converted as a part of the answer returned from the look-up table. Then, if a conversion is required , we can insert a new operator into the expression, a "convert from old type to new type" operator that acts as a unary operator during run-time evaluation.

In Figure 2-1 we view the expression as a binary tree before and after the conversion is added. These inserted type conversions can then be used to bubble-up the result **type** of the tree during type evaluation of more complicated expressions, as they inherit the type resulting from the symbolic application of the conversions to their children. During value evaluation, the inserted conversions are applied to their children and the resultant **value** is passed up the tree.

In the abstract, type evaluation consists of recursively (1) traversing through an expression until an operator is encountered, (2) locating the associated operand(s) of that operator, (3) symbolically evaluating this sub-expression through some mechanism that returns a type and identifies required run-time conversions (or detects type consistency errors) and (4) inserting these conversions in the proper location within the expression.

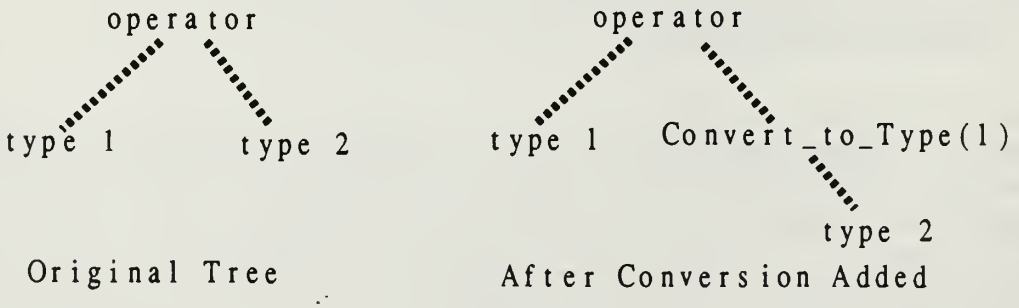


Figure 2-1. Type conversion inserted into tree

III. PROGRAM DESIGN

A. DESIGNING FOR CHANGE

1. Design Principles

Because we have chosen the program family approach to the design, we are planning for inevitable changes from one instance of the program to another. We use information hiding and modularization as the basic principles of our design to help plan for change. By isolating the changeable parts into modules and defining module interfaces that are insensitive to change, the problems associated with transforming a program from one application to another can be minimized [6]. That is, by encapsulating secrets into modules, restricting the information distribution between modules, and viewing the program as a set of modules that have well defined interfaces, we design in flexibility that is not found in other designs (like, for example, designs that are composed of a chain of data transforming components). The other critical principle we follow is to put off the design of the data structures as long as possible. In this way we remain as close as possible to the abstraction of the problem during design, and can better capture the key algorithms of the process without needless detail.

2. Identifying the Changes

Most programs undergo changes during their life cycle, whether these are modifications during the design phase, post-implementation upgrades, or extensions to original capabilities. By deciding to use the program family approach, we desire to design flexibility into our design in the sense of allowing ease of modification and

extension from one application to another.

Those modifications and extensions we are primarily concerned with are the different data structures which support type information from one application to another, the different actions of the operators that perform evaluation of these different type data structures, and the environment in which the expressions may be found. Another view of the problem may focus on the environment in which these expressions are being evaluated in. Clearly, the evaluation process does not change with the environment, but the types and the operators do. That is, the environment in which an expression occurs changes. If we are assuming that the environment will change from one application to another, we want to encapsulate that change through modularization [7]. For example, one application may use the simple arithmetic operators to evaluate an expression of types real and integer. For an application that must perform type evaluation of matrices whose entries are also reals and integers, we have a set of operators for matrix arithmetic. Another application may combine sets of record types, with the fields of the records composed of boolean, numerical, and string types, with set operators. Type evaluation in this environment may involve a determination of equality of members of the sets, in which case the types of each member must be compared. So the environment will change from one application to another as the language or system in which the expressions are composed changes. In one instance the expressions may be the right hand side of an assignment operator, as is found in most programming languages. Another may represent the validation of the result type of a function in a modeling language. In any case, the environment must be constructed in which the type evaluation of the expression is valid. This means defining the allowed operators, constants, variable

declarations, parameters, functions, and initializing them as necessary. We need a core program that can easily be modified to different environments.

B. PROGRAM DESIGN

1. Creating the Environment Through Declarations

In any environment, we say that an object has a value and it has a type. Type evaluation of an expression involves getting the types of the objects in the expression, which means the object must be known to the environment. This presents several issues:

(1) How does the environment know about the objects?
(2) When are the objects assigned type?

For (1), we say this action occurs during the declaration phase, in which statements indicating what kind of object is declared are made to the system in the language of the environment. For example, in Pascal, there are constant, type, and variable declaration blocks in which a user may enter the information to identify objects of these kinds. In a spreadsheet, a user fills in blocks or whole lines and columns with names and values or predefined system functions such as sum, division, and total (for a column of figures). Modeling languages also use statements for declaring objects to the system, such as functions, constraints, or parameters. Since this declaration phase is common to all these systems, we want to include this capability in our program family. As for (2), type assignment may be static, with the type declared during object declaration, or it may be dynamic in the sense that a declared object may be assigned the type resulting from the type evaluation of an expression. For example, in spreadsheet applications, a block may be identified to be the sum of an entire column of figures. The type may then be determined during evaluation of the sum operation and assigned to that block. In

languages, the modeler may declare what the type of a function should be, and then use the resultant type from the evaluation of the function to validate the declared type. We may also encounter mixed systems, in which the typing of the data is mixed with the model. For some objects in the model the type may be part of the data and thus only available at run-time. We consider only static typing, in the sense that the type of all objects is assigned prior to run-time and these type assignments are not changed dynamically during execution, since this is the common capability in most languages. From these basic requirements, a user model is defined that satisfies them, as depicted in Figure 3-1. It takes an input source that defines the environment through declarations, performs environment declared functions on objects in the environment, and returns the results. For our research, we are concerned with expression evaluation as an environment function. Since the validity of the expression depends on the language of the environment , we first focus on the change in language from one environment to another.

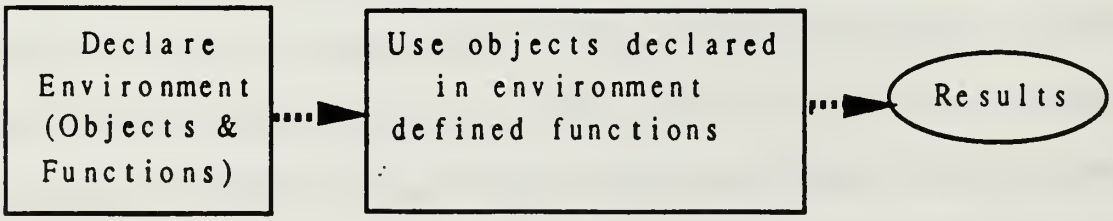


Figure 3-1. User Model

a. Interpreting the Language

A valid expression is composed of declared objects , but these objects must be identified in and extracted from some source language. If we consider the environment to be defined through a language, then the declarations phase can be thought of as an interpretation of the source language. In compilers this interpretation is done via a sequence of steps:

a. lexical analysis
b. syntactic analysis
c. semantic analysis/processing
d. storage allocation
e. target code generation
f. assembly

We borrow the first three steps, replacing the last three with operations and functions necessary to achieve the desired results. The lexical and syntactic analysis will constitute the recognition phase, while the semantic analysis will be type evaluation. Whereas a compiler produces intermediate code that reflects the types of operations available in a specific machine [3], our semantic process reflects the mechanisms available for type operations in a *specific application* and produces the type resulting from the symbolic evaluation of an expression.

b. Lexical Analysis

The primary task of lexical analysis is to assemble characters from the input stream into tokens, and to determine which terminal symbol of the language each token is an instance of [3]. The terminals are classified for this purpose into *lexical token types*. Since our approach is to build a program family, and the language (environment) of the

expressions from one family member to another may change, we build a *generic* lexical analyzer module, which may be accessed by application specific syntactic analyzers. It is generic in the sense that it always return a scan token from the same set of possible "generic" scan tokens, regardless of the environment. The set of possible generic scan tokens may be defined over a domain of input that is common to all applications, and these become the building blocks of application specific tokens. Since the lexical analyzer will not change, it becomes one of the program family core elements.

As most languages consider identifiers to begin with an alphabetic character, and real and integer numbers to be recognizable objects, we will include these as generic token types. Lanugages differ in interpetation of the various other printable and non- printable (control) characters, so these will be considered as special tokens, with the exception of "end of line" and "end of file". These are used as indicators in most languages so it seems natural to identify them as separate objects. We can then define the ASCII character set to be the domain of our generic scanner function, and the range to be as follows:

(1) An identifier token begins with a character in [A..Z,a..z]
ends when the next character is non-alphanumeric

(2) An integer consists of a string of numerals and ends
when the next character is non-numeric

(3) A real consists of a string of numerals, a decimal,
followed by one or more numeric characters and ends
when the next character is non-numeric

(4) Blanks

(5) The "end of line" token

(6) The "end of file" token

(7) All other ASCII characters are special tokens

For consistency, we also include an "unknown" token for error handling.

A scanner may need to look ahead in the input stream while building tokens (for example, to determine when it has seen the last character of a token), so we use a buffer to hold the contents of the input. The buffer itself can also remain unchanged from one family member to another and we need to have certain buffer functions to access it without errors and support the lexical analysis. The following are then provided as elements of the buffer module:

procedure Fill(Buffer):
Prepares input to be accessed via buffer operations

function Empty(Buffer):
Returns status of buffer

function LookAhead:
Returns next character without consuming it

function GetChar:
Returns next character and consumes it

Any environment specific syntactic analysis that requires a token must go through the generic lexical analyzer, and we define the interface to be:

procedure GetScanToken:
Builds one of the generic tokens by using buffer operations

By using a generic lexical scanner, we lose some of the processing power of a fully

specific one, so we shift some of the work to the application specific syntatic analyzer. For example, the scanner may return the character "@", which is a special token as identified by the generic lexical analyzer, followed by an identifier as two separate tokens. If the environment recognizes identifiers that begin with "@" to be declared a certain type of identifier, then the specific syntactic analyzer would combine the two into one token prior to using it.

c. Syntactic Analysis

While every environment may have differences in typing information representation, operator naming, and the expected order in which environmental information is encountered in the input stream, we assume that in all of the environments one or more of the following occur:

- objects such as constants, variables, types, and parameters are declared in the environment prior to use
- the types of the objects can be assigned during the declaration phase
- or, the type of an object may be assigned by evaluating an expression
- or, the type may be assigned after evaluating an expression and then comparing that type to the expected type

During syntactic processing we need to determine which kind of declaration is to be made according to the stream of tokens. This is performed by recognizing sentences in the language of the environment. In compilers, this language recognition occurs through a syntactic analyzer that operates based on a fixed language. Since we expect the language to change and we already have a generic lexical analyzer to get the

tokens for use in this recognition process, we may look at which functions are common to all sentence recognitions. We can make these functions individual, generic pieces of the program family syntactic analyzer. Then, we can tailor make an environment specific analyzer through the ordering and repetitive use of these functions to recognize the language.

First we note that in any sentence a language recognizer may do several things upon encountering a token from the lexical analyzer:

- it may recognize it as valid and consume it, possibly causing an action to occur
- it may use it to construct an environment specific token, which also consumes it
- it may ignore it and look for the next token
- it may have expected a different token, causing either a warning or error to be produced

From one language to another we want a function that can do these actions based on requests from the current state of syntactic analysis. We define the function

GetNextScanObject(Req1,Req2,Req3,Req4,Req5,Req6,Req7,Req8)

that based on a request being either Yes (for get the object), No (for ignoring the object), Warn (for generating a warning), and Err (for generating an error and halting the scan process), can be used to return any of the following eight generic tokens:

1. Identifiers
2. Real Numbers
3. Integers
4. Special Characters
5. Blanks
6. End of Line
7. End of File
8. Unknowns

For example, a call such as

GetNextScanObject(Yes,Yes,Yes,Warn,No,No,Err,Err)

would return the next identifier, real number, or integer returned by the lexical analyzer, ignoring all blanks and end-of-lines. Warnings would be issued if any special characters were encountered during the process, while the end-of-file or unknown token would cause an error message and system halt. Another common feature in sentence recognition is that the validity of the current token in the sentence depends on what the next token is. The function **LookAheadNexScanObject** returns the next token without consuming any tokens. This token can then be compared to what the recognizer expects, and that result used to control the recognition process.

Since we have said that the generic token types include those kinds of tokens recognized by most environments such as identifiers, real numbers, and integers, but also a group called special tokens which will be viewed differently in the various environments, we need a common way to detect whether the current special token is one expected, or to use it in syntactic control. We use the function

that takes as its arguments the current token and the environment expected token, returning a boolean response to the comparison. All of these functions then become core elements for the program family syntactic analysis phase, and it depends on the environment as to how and in what order they are used.

The next step in the recognition phase is to determine what action takes place after a sentence is recognized. Since the recognition of a valid sentence must result in a declaration, this action must install the objects in the environment for possible later access. If we consider the varied environments, it could be a variable or parameter being declared, it could be a function declaration, or any object in the environment. For each of these, the recognition must result in the object being stored through some semantic action. Our semantic action will cause the object to be stored in a symbol table. Initially, we will have the name of the object, but each object has other attributes as well. The attributes of each object include:

- its name
- its type
- its value (numerical, boolean, other)
- the kind of object it is (eg operator, operand)

The type assignment of each object will occur at some point in the process, so this is a common function. Since each environment will use different types and different data structures to capture the type information for inclusion into the symbol table, we need a

procedure that reflects this common function in all the environments but hides the specific typing details. We then define the function name and interface to be function **GetTypeInfo** that will syntactically analyze the typing information. It will return the kind of object recognized, its value, create the data structure for the types allowed in the environment, and return this information. Since the typing information will vary from environment to environment, we return only a pointer to the typing data structure. The object and its attributes must be put into the symbol table so we provide the function **Insert** which creates a symbol table entry and installs the object into the table. The table must also be initialized, which includes creating it and filling it with environment specific operators. The procedure **Initialize(Symbol_Table)** is defined by name and argument, but will be specific to each environment. At this point we can identify the lexical and syntactic analyzer modules, as well as the symbol table module in Figure 3-2.

2. Type Evaluation Phase

   a. A Generalized Expression Grammar

Having considered that the environment of the expressions changes, we also want to incorporate flexibility in terms of changing types in our design of the type evaluator. By viewing it from a user's point of view, we can determine what it is that is changeable from one user (environment) to another. We say a user has one or more expressions and wants a program that evaluates them and returns their types, or detects errors. Expression (operand) types and operators may include:

- unary or binary
- arithmetic
- boolean
- matrix/vector
- interval arithmetic

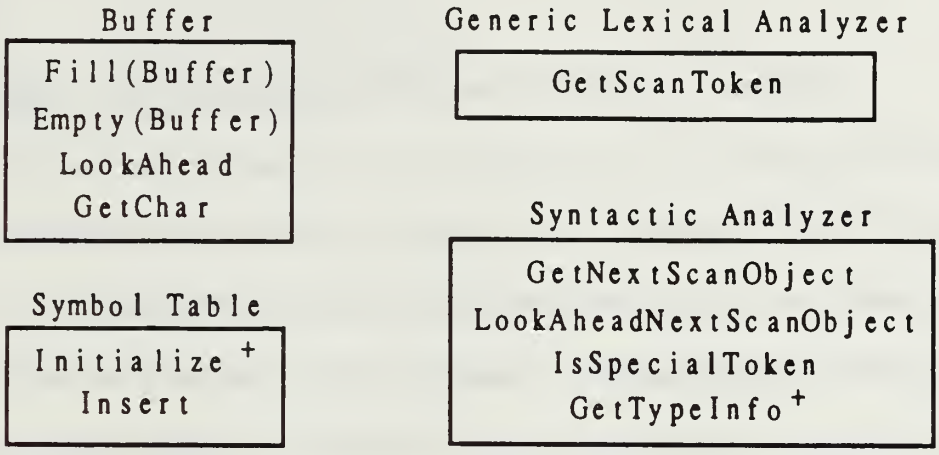


Figure 3-2. Elements of the Program Family

- dimensional units
- others as defined by the user during declarations

Errors detected include:

- incorrect operand type matched with an operator that does not operate on that type
- missing operand(s)
- missing operator
- incompatability between the types in two sub-expressions

We expect the types and operators will change, but not the error handling requirements. To detect errors, compilers use parsers to recognize a valid sentence in a grammar. If we call expressions the sentences of our language, then we need a grammar that recognizes them.

We have said that the process of type evaluation will not change from one environment to another, yet the operators may. We need to hide this change in our grammar. From our abstraction, the operators have precedence and degree. We may define a grammar that captures the precedence and degree of operators, but in a way that preserves information hiding. If we assume that all the operators in the problem domain are either degree 1 or degree 2 (that is, unary or binary), and that they can be placed in a finite set of precedence levels, we can define a grammar for expressions in general. For instance, unary -, unary +, "NOT", "sum", "sin", and "log" are all unary operators and can be placed in a class we term UOP. The usual arithmetic operators "+" and "-" are placed in the class BOP1, as well as the boolean operator "OR". Multiplication and division are in BOP2, since their precedence is lower then addition and subtraction, along with "AND", since its precedence is lower than "OR". The relational operators "=", '<', '>' and so on are in the class EQOP, which have the lowest precedence. By lowest precedence we mean they are applied after all other operators. For example, our precedence classes could be:

UOP : NOT, unary -,+, Sqrt, Sin, etc
BOP1 : *, /, Mod, And
BOP2 : +,-, OR
EQOP : =,<,>,<=,>=

We then provide one production rule for each level of precedence and terminal and write the grammar as:

```
Exp       ---> Term (EQOP Term)*
Term      ---> Factor (BOP2 Factor)*
Factor    ---> Primary (BOP1 Primary)*
Primary   ---> Element | UOP Element
Element   ---> Operand | (Exp)
```

This becomes a generalized grammar for recognizing an expression [4].

Grammars of this form that are left-factored and have no left-recursion are suitable for use in a top-down parsing technique, so we use recursive descent. Each production in the grammar is transformed into a recursive procedure, and by including semantic actions at appropriate places in the code we transform the expression from an infix list to a postfix tree, suitable for use by the type evaluator. To include more levels of precedence we would define each level as a new production in the grammar and then write it as a recursive procedure. We call this the expression parser, and it will be another core routine of the program family.

b. Symbolic Type Evaluation and Conversion

Now that we have the expression represented as a binary tree with the root node and all internal nodes as operators and all the leaves as operands, we want to symbolically evaluate it. This is the same as getting the type of the root node, which depends on the types of all its children. Not only do we want to get this type, but we want to insert any allowed type conversions into the tree in the proper position so that during actual value evaluation they will be handled in the correct sequence. We can then say that to perform type evaluation/conversion, we need three things: an algorithm for

traversing the tree, a mechanism for determining the types of the inner nodes depending on the operator at those nodes, and an algorithm for inserting conversions.

We implement the traversal algorithm as a function that returns the type of the root node of the postfix tree. In order to get the type of the root node, it must recursively descend the tree, get the types of all sub-expressions, and bubble up the result. If the expression is simply an operand, the root node has no children and its type is obtained from the symbol table. If the expression is a unary operator and an operand, the root is the unary operator and it will inherit the type of its child. Similarly, if the expression is more complex, the algorithm must determine the types of its right and left children, then, depending on the kind of operator that is the root node, the type of the root node is determined by symbolically evaluating this sub-expression. That is, there will be different mechanisms for determining the type of the root and all inner nodes depending on whether the operator is multiplicative, additive, boolean, or other, and on the types of the children (operands). But these mechanisms do not affect the traversal algorithm, as it is only concerned with whether the root node is an operand, a unary, or a binary operator. To implement this traversal algorithm, we use a recursive procedure **GetType** with the following design features:

- the argument to GetType is a pointer to an object
  in the tree, which is in the symbol table
  (since all operands and operators are in the table)

- this argument is checked against a case statement
  to determine how to continue traversing the tree
  and when to call the look-up mechanism, based
  on the kind of object as:
  Case object of:
  operand : get the type from the symbol table
  unary op : 1. get the type of the right child
  using GetType
  2. call look-up function to
  symbolically evaluate operator
  binary op: 1. GetType of right child
  2. GetType of left child
  3. call look-up function to
  perform symbolic evaluation

- If it's an undefined type, meaning it is not found in
  the symbol table, a nil pointer is returned

Since the algorithm does not depend on what specific kind of operation is involved, for instance ,+ or *, rather on whether it is an operand, unary operator, or binary operator, the algorithm will remain unchanged from environment to environment.

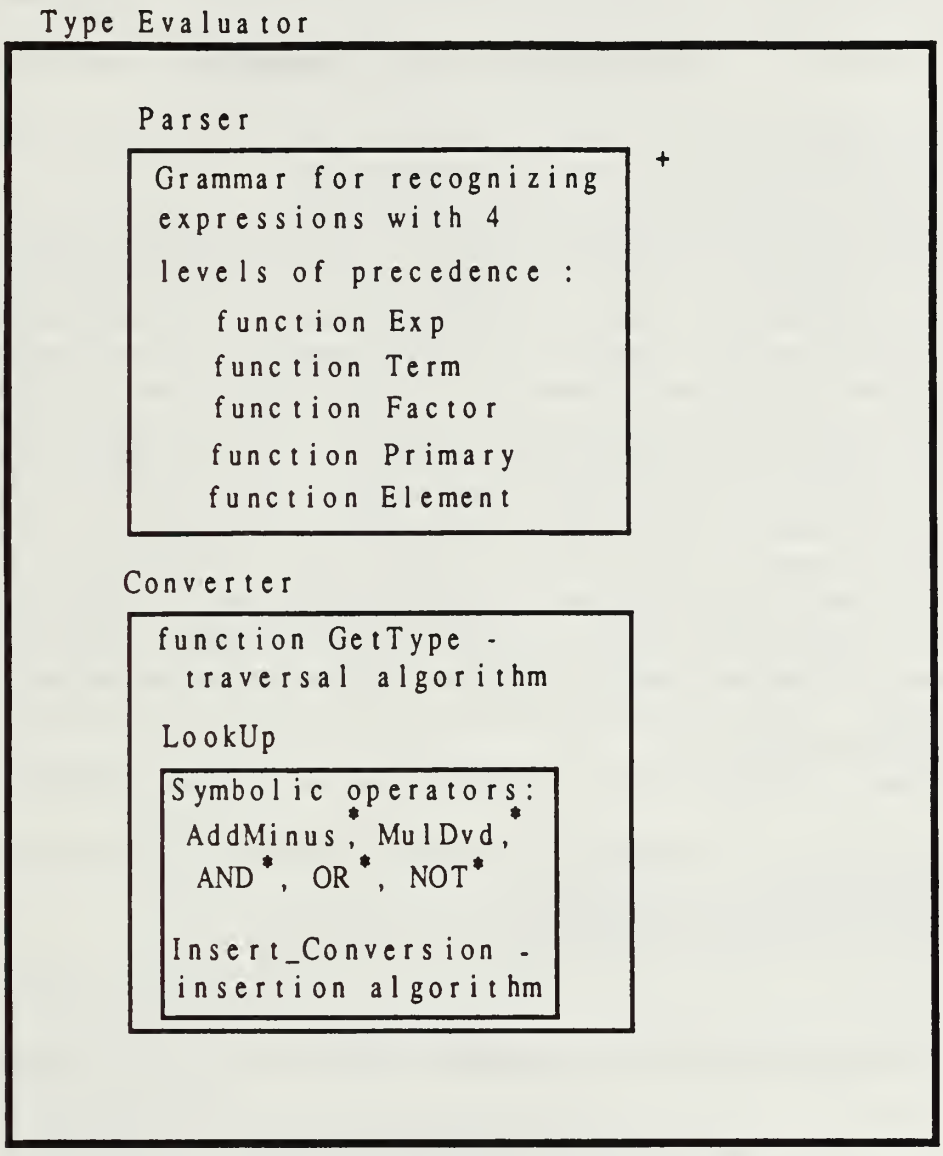
During this traversal, calls are being made to a look-up function in order to symbolically evaluate sub-expressions. What changes during these calls are the symbolic evaluation mechanisms in function **LookUp,** as these have knowledge of the environment specific types and operators. The functions that these mechanisms perfom will be the same:

(1) determine if the operator and operand(s) are consistent
(2) return the result from symbolically evaluating the
operator and operands
(3) identifies which, if any, of the operands
require conversion

One method of look-up mechanism is to use tables, with one table for each operator and the types as indices into the table. Rather than determine all possible operator and type combinations , which would result in large and complicated tables of different sizes, we use instead routines that explicitly check for type validity, identify the conversions required, and return the result of the type evaluation based on the operator. Because a member program will be designed for a particular environment, the allowable type conversions will be known and can be written into these routines. By specifying only the interface to these routines, we preserve infomation hiding and allow the internals of the routines to be designed as the user desires. In the function **LookUp** , we include a case statement such that for the recognized operator, the environment specific evaluation routine (mechanism) is called. As an example, we need a different evaluation routine for the types real & integer, string, and array for the additive operator "+".

As the traversal algorithm in **GetType** is indepedent of the specific operator types, so is the insertion algorithm in function **LookUp.** We implement this indepedent process as procedure **Insert_Conversion** , that takes as its' arguments the left and right operands and the identified conversion operator and inserts it in the correct branch of the tree.

We can now identify the type evaluation module and identify those parts that are considered core routines for the program family and those that are environment specific in Figure 3-3.



+ - Indicates addition/deletion of recursive functions to add/delete levels of operator precedence

* - Indicates routines modified to handle environment specific type data structures

Figure 3-3. Program Family Core for Type Evaluator

IV. EXAMPLE FAMILY MEMBERS

A. REAL/INTEGER/BOOLEAN AS A PROTOTYPE

From the design of the program family, we have noted that to create a specific member of the family, we need only modify the core in certain areas and add specific routines. We must declare the operators and the types, and then we must include routines for these operators that determine the resultant type after performing the symbolic evaluation as well as which, if any, of the operands need conversion. As a prototype member of the program family we consider an application that includes the common arithmetic and boolean operators and types. We then define the operator set for this instance of the family to be:

```
ExpressionOps =
 (uminus,uplus,plus,minus,dvd,mul,
  AND,OR,NOT,EQUALS,Not_EQUALS,GTE,LTE,GT,LT)
```

These operators are then assigned a precedence class within the set of expression objects as:

```
ExpObjects = (UOP,BOP1,BOP2,EQOP,OPND,LftPARAN,RtPARAN);
UOP  = (uminus,uplus,NOT);
BOP1 = (mul,dvd,AND);
BOP2 = (minus,plus,OR);
EQOP = (EQUALS,Not_EQUALS,GTE,LTE,GT,LT)
```

The allowed types in this application are then Integer, Real, and Boolean. By using a pointer, we have put off the decision as to what data structure we want to use for the type until implementation of a specific member. For this example, we will use a single character to signify the type, with "I" representing integer, "R" the real type, "B" boolean, and "U" the unity type (for the arithmetic operators, since they are overloaded). While we allow expressions that mix the types consistent with the correct operators, the only type conversion that can take place during type evaluation is **Integer-to-Real.** Although sub-expressions such as "6 < a", where "a" is of type real are allowed, no conversion is identified by the routine that perfoms type evaluation for the relational operators. Instead, it always returns a boolean type, but will identify any **Integer_to_Real** type conversions needed. These conversions are inserted into the expression as a unary operator. Both of these processes take place during calls to **GetType** , which uses the function **LookUp.** The routines we provide in function **LookUp** to perform type evaluation for the set of operators declared in this application are:

uplus,uminus: U_AddMinus
Returns the type of the right child

plus,minus : AddMinus
A routine that takes the types of the left and right operands as input and determines (1) that they are of type integer or real and (2) if one is integer and one is real, identifies that operand for conversion.

mul,dvd :
Since integer and real arithmetic always results in a real or integer type, AddMinus is also used for these operators.

NOT : NOT_
A unary operator whose only legal argument is boolean and whose result type is always boolean; identifies no conversions

AND,OR : AND_OR

Both of these are binary boolean operators whose only valid arguments are boolean, so it checks the consistency of the operands and returns boolean or error as type.

Not_EQUALS,
EQUALS : EQUALS_

This operator may have boolean or arithmetic types as operands in pairs, thus it must check for correct operand types, and for integer and real, it identifies any conversion required.

GTE,LTE,
LT,GT : RelOPS

These are all relational ops that apply only to integer and real types, so it performs the same function as AddMinus, but returns a boolean type.

The only modification we need make is within the function LookUp we insert the following:

```
Case Operator of
 uplus,uminus : U_AddMinus(Rt,Conversion);
 plus,minus,mul,dvd : AddMinus(Lft,Rt,Conversion);
 NOT : NOT_(Rt,Conversion);
 AND,OR : AND_OR(Lft,Rt,Conversion);
 EQUALS, Not_EQUALS : EQUALS_(Lft,Rt,Conversion);
 GTE,LTE,GT,LT : RelOPS(Lft,Rt,Conversion);
end;
```

where Rt and Lft are the types of the operands, and Conversion is the record that holds the conversion identification information. Since the insertion algorithm depends only on internal identification of which, if any, children of an operator node need conversion, this will remain unchanged.

We add an output routine to write out the typing information for this specific type data structure and our program is complete. As an example, suppose we want to evaluate the expression

A AND NOT (b = c - d) OR E

where A and E are boolean types, b & c are integer types, and d is of type real. The expression is formed into a linked list during the declaration phase as :

A: Boolean -->
AND : Boolean -->
NOT: Boolean -->
(: Unity-->
b: Integer -->
EQUALS : Boolean -->
c: Integer -->
- : Unity-->
d: Real -->
): Unity-->
OR : Boolean -->
E: Boolean -->

This list is sent to the type evaluator where it is parsed according to the precedence grammar rules into a binary tree in postfix form, with the root node the operator of lowest precedence as in Figure 4-1. The parsed postfix tree is then sent to the conversion module, which gets the type of the root node through recursive descent, calling the appropiate conversion module depending on the operator which makes up each inner node. After the look-ups are completed and the insertions are made, the type evaluator returns the type of the root node, and the postfix expression now looks like Figure 4-2. During actual evaluation, the conversion **Integer-to-Real** is handled as a unary function, converting an integer number into the machine specific real version. We note that no modifications were made to the algorithms which perform the symbolic type evaluation.

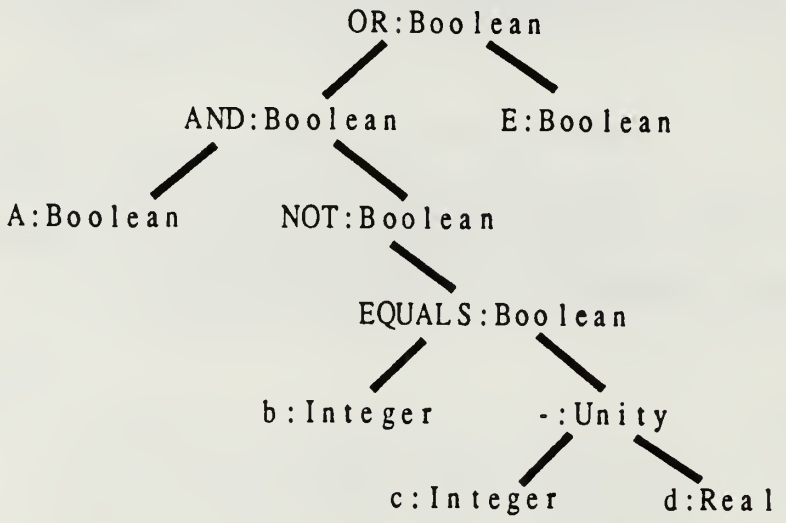


Figure 4-1. Expression as Postfix Tree

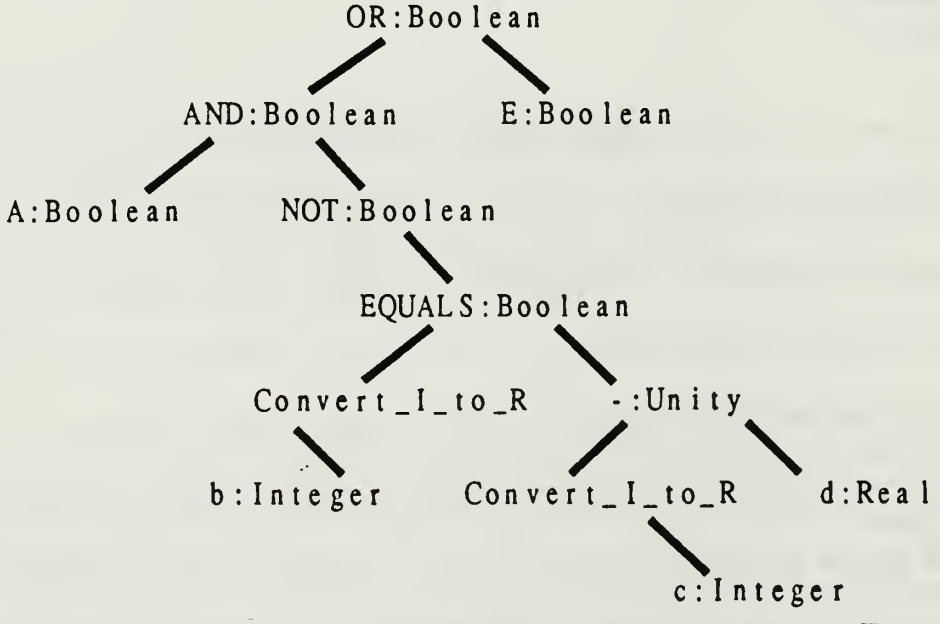


Figure 4-2. PostFix Tree with Conversions Inserted

B. INTERVAL ARITHMETIC

1. Intervals as a Type

Another set from the problem domain of expressions found in the various environments is the set of all expressions involving interval arithmetic. We define an interval to be a set of real numbers [a,b] such that a <= b. The operations on intervals A = [a1,a2] and B = [b1,b2] are then calculated explicitly as:

A + B = [a1+b1,a2+b2]
A - B = [a1-b1,a2-b2]
A * B = [min {a1b1,a1b2,a2b1,a2b2},max{a1b1,a1b2,a2b1,a2b2}]
A / B = [a1,a2] * [1/b1,1/b2]

Problems in interval arithmetic may be divided into two general classes: problems with "inexact" and with "exact" initial data. The first class involves problems of the kind that involve solving equations (expressions) whose data are allowed to vary over an interval by computing a best possible inclusion set. An example from Gotz [8] of this kind of problem is a set of linear equations whose co-efficients are allowed to vary over a set. In the class where exact data are given, the interval analysis is mainly used to develop methods that generate convergent sequences of bounds converging to the solution under comparitively weak conditions. For example, if we have a polynomial p(x) in the form

$$p(x) = (((...(a10x + a9)x + a8)... + a1)x + a0$$

and we consider the right hand side to be an expression we want to evaluate in the interval sense for the coefficient intervals a10 = 10, a9 = 9,..., a1 = 1, and a0 = [-0.02,-0.01], x = [-0.1,0.02]. We wish to determine whether 0 is in p(x), that is, we wish to

determine the "type" of the resultant interval for the initial coefficient values and intervals, without explicitly solving the equation for x to find p(x) = 0 [8]. To do this would require numerical evaluation of the equation using some standard numerical method, such as Newton's method or the Runge-Kutta method, both of which perform repeated iterations using approximations that are closer and closer to the real solution as input for the next iteration. To solve problems of this type, we need to implement the interval operations and integrate them into our type evaluator design. If we say that operands have intervals as types, then we can have the type evaluator return the resultant bounds or type of an expression. Such an expression type evaluation may be part of a two step process for the realization of machine interval operations as discussed in Gotz [8] :

> (1) a logical part which determines the bounds of the result of combining operands which can be calculated according to the defined operations of interval arithmetic;
> (2) an arithmetic part where the pair of bounds of the resulting interval is calculated with the help of a directed rounded machine interval arithmetic that keeps track of precision.

In (2), round-off errors need to be considered as well as instances where the numerical answer is zero, if this is a consequence of the machine the program is running on and not the operator. For interval arithmetic, we could have routines that pre-determine whether the result of an operation will exceed the maximum allowed number for a machine. We do not consider precision here, however we may implement the function of (1) using our core program.

2. Implementing Intervals

We will allow the same arithmetic operations as for the previous example, including unary plus and minus. The members of the precedence classes are:

```
UOP  = (uminus,uplus);
BOP1 = (mul,dvd);
BOP2 = (plus,minus)
```

The others have no operations in this environment and are not used. We perform the declarations as before, and fill the symbol table accordingly. As before, each entry in the table has a pointer to its type. This time we define the type to be a record with two fields, "left" and "right", both of which are of type real. By changing the pointer type definition in the declarations, all other code remains the same since it references the pointer only, not the type it points to. Besides this change, the only other modifications we need are to the routines in function LookUp that determine the resultant types according to the current operator and the output routine. Thus, we include in function LookUp the type evaluation functions:

```
Interval_Unary_Minus:  Returns the negative of the interval as
                       the type
Interval_Unary_Plus:   Returns the interval of the operand
Interval_Add:          Performs actual calculations based on the
                       interval operators and returns an interval
Interval_Sub:          Same
Interval_Mul:          Same
Interval_Dvd:          Same
```

The section of code in LookUp that calls these functions now becomes:

```
Case operator of
 uminus: Interval_Unary_Minus(Right,Conversion);
 uplus  : Interval_Unary_Plus(Right,Conversion);
 plus   : Interval_Add(Left,Right,Conversion);
 minus  : Interval_Sub(Left,Right,Conversion);
 mul    : Interval_Mul(Left,Right,Conversion);
 dvd    : Interval_Dvd(Left,Right,Conversion);
```

As an example, consider the expression

a / (b - c) * d

where the initial data is:

a : [1.75,-1.50]
b : [1.20, 0.32]
c : [2.30,-1.20]
d : [1.40, 0.05]

After the expression is formed into a linked list during the declarations phase it is sent to the type evaluator where it is parsed into its postfix binary form, as in figure 4-3. During symbolic type evaluation, actual numbers are being calculated, but they are the type of

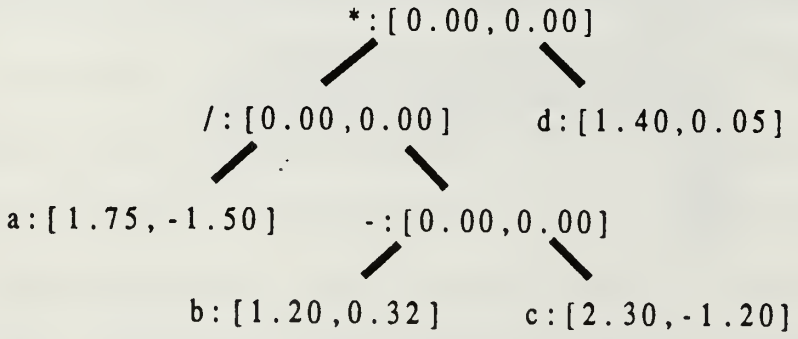


Figure 4-3. Interval Expression as Postfix Tree

the operands, not the value. The resulting tree then looks like Figure 4-4. The resultant type is then the value of the root node, [-1.68,-0.06]. During the type evaluation the inner nodes appear to be assigned a type other than unity ([0.0,0.0]). Because the function GetType recursively returns a temporary variable as the type , this assignment is only made to this variable, which is bubbled up the tree. The symbol table entries for any objects are not changed during the process.

C. THE DIMENSIONAL MODEL TYPE

1. Description of Operations Research Model

As a more powerful example, we consider the user of type evaluation in an executable modeling language. As discussed in [1] and [9] an executable modeling language (EML) would provide an alternative to the usual method for constructing and executing models, which is to have the modeler describe the model in an informal algebraic notation, then develop a matrix generator computer program to construct the problem in the form required by an optimizer system. The EML approach would allow a

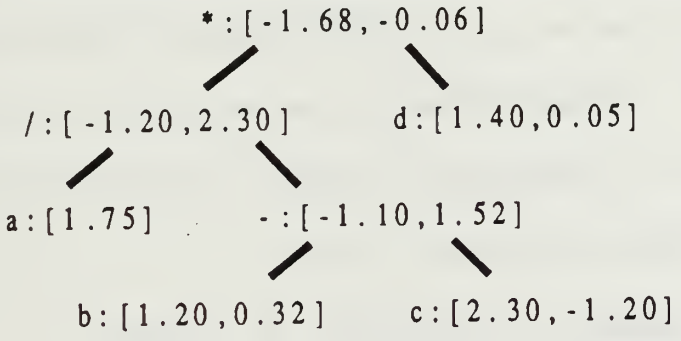


Figure 4-4. Interval Postfix Tree After Evaluation

modeler to conceive, record, and validate his model using a modeling language with algebraic notation that also documents the model. The model in this notation is read by a computer, translated into a form for optimization, solved, and its solution returned for analysis -- all without manual intervention [1]. A "type" system for objects in the model can be designed and the system can be added to any executable modeling language. Such a system, composed of the definitions of the types allowed in the model together with the calculus for operating on the types, allows type validation of a user's model.

As an example, [1] specifies a type calculus for an extended dimensional system that determines if the model is well formed in the sense that functions and constraints consist of homogeneous components. Each variable, constant, function, and constraint in the model is assigned a type that consists of its "concepts","quantities" and "units" of measurement.

The modeler assigns each numeric valued object a type that consists of a concept description, quantity description and unit description. The concept represents the essence of the object, examples are BUTTER, GUNS, and TIME. A quantity is a measurable attribute of the concept, such as WEIGHT, COST, DURATION, VOLUME, and CARDINALITY. The units are from specified unit systems with conversion factors between units, such as [INCHES,FEET,YARDS], [OUNCES,POUNDS,TONS], and [HOURS,DAYS,WEEKS]. An example of a type is:

WEIGHT	of	@BUTTER	in	POUNDS
quantity		concept		unit

where concept identifiers are prefixed with "@" to avoid confusion with quantity identifiers. During type validation, an EML can do automatic conversion of units,

manipulate scale factors, and apply user supplied concept, quantity, and unit conversions. The type system also allows a hierarchy of concepts and provides for inheritance of quantities. This type of function is what the modeler needs to verify the model, however, during actual execution of the model the inserted conversions must actually perform the necessary numerical computations when converting from one type to another. As one of the functions that is specific to any particular application because it has knowledge of the application types, **ReturnConvFactor** takes as its arguments the type to be converted and the target type it is being converted to and determines the composite numerical conversion needed to reflect the conversion of the value of the converted type. The modeler defines the allowed conversions and provides the numerical conversion values for these allowed conversions in the function. Since the look-up routines determine if the resultant type evaluation between operators and operands is valid or in error prior to calling **ReturnConvFactor**, only valid conversions are expected. During type evaluation of the expression, the conversion factor then becomes the value attribute of the conversion node. We then apply this value during value evaluation to the child of the conversion, with multiplication as the operator.

2. Model Example

As a specific example, consider the following text file, which may be a part of the complete specification for a model. Within each declaration block, the typing information for an object is enclosed within "<< >>".

```
SETS
  DAIRIES i; <<nominal>>
  WAREHOUSES j; <<nominal>>
  PATHS(i,j) := {DAIRIES} x {WAREHOUSES};
VARIABLES
  SHIPMENTS1(i,j) {PATHS};  <<WEIGHT of @BUTTER/
    DURATION of @TIME # POUNDS/HOUR #>>
  SHIPMENTS2(i,j) {PATHS};  <<WEIGHT of @BUTTER/
    DURATION of @TIME # TONS/DAY #>>
PARAMETERS
  SCOST(i,j) {PATHS}; <<COST of @OBJECTIVE/(WEIGHT of @BUTTER/
     DURATION of @TIME) # DOLLARS/(POUNDS/DAY) # >>
  SUPPLY(i) {DAIRIES};<<WEIGHT of @BUTTER/DURATION of @TIME
     # POUNDS/DAY # >>
  DEMAND(j) {WAREHOUSES}; <<WEIGHT of @BUTTER /
     DURATION of @TIME # TONS/DAY #>>
FUNCTIONS
  OBJECTIVE := SUM(i,j){PATHS} ((SHIPMENTS1(i,j)+SHIPMENTS2(i,j))
    *SCOST(i,j));
         <<COST of @OBJECTIVE # DOLLARS# >>
CONSTRAINTS
  OUTBOUND(i) {DAIRIES} := SUM(j) {PATHS}
                  ((SHIPMENTS(i,j)) =L= SUPPLY(i));
    <<WEIGHT of @BUTTER/DURATION of @TIME)
    # POUNDS/DAY) # >>
  INBOUND(j) {WAREHOUSES} := SUM(i) {PATHS}
                  (SHIPMENTS(i,j) =E= DEMAND(j));
    <<WEIGHT of @BUTTER/DURATION of @TIME # POUNDS/DAY # >>
```

Note: This implementation does not contain indexing and
set (SUM) operations

During type validation, the type system symbolically evaluates the function OBJECTIVE and then checks to see if the result type is the same as the modeler intended. The right

hand side of the function OBJECTIVE is then treated as an expression. In the first step the expression is created as a linked list. Next a pointer to the expression is passed to the type evaluator and the expression is parsed into its postfix format. The expression now looks as depicted in Figure 4-5(a). In the second phase of type evaluation the actual symbolic evaluation, including insertion of appropiate conversions, results in the tree of Figure 4-5(b). While the system has inserted allowable conversions during the type evaluation, the modeler is not concerned with this, but rather the resultant type. Now that the type evaluation has been performed and a resultant "type" returned as an answer, it is up to the modeler how to use this information. During model validation this information can be used in the following way: This resultant type is used as an operand to build an expression that is also type evaluated. An object in the environment such as a function or a constraint may have a type declared which is actually in the symbol table. This type becomes a second operand and the equality operator "=" is the expression operator. During type evaluation of this expression, the model object can be called type valid if there are no conversions inserted into the symbol table type, only the result type of the initial expression.

In another model, the resultant expression can become the declared type of an object, which may itself be used later in another expression. By providing the modeler with the utility of type evaluation and the non-changing lexical and syntactic routines to custom design the semantic actions that are the control flow for the model environment, flexibility in designing a working model makes the task much easier. The only constraint upon the modeler is that an object referenced during expression evaluation has been assigned a type; when and how the type is assigned is up to the modeler.

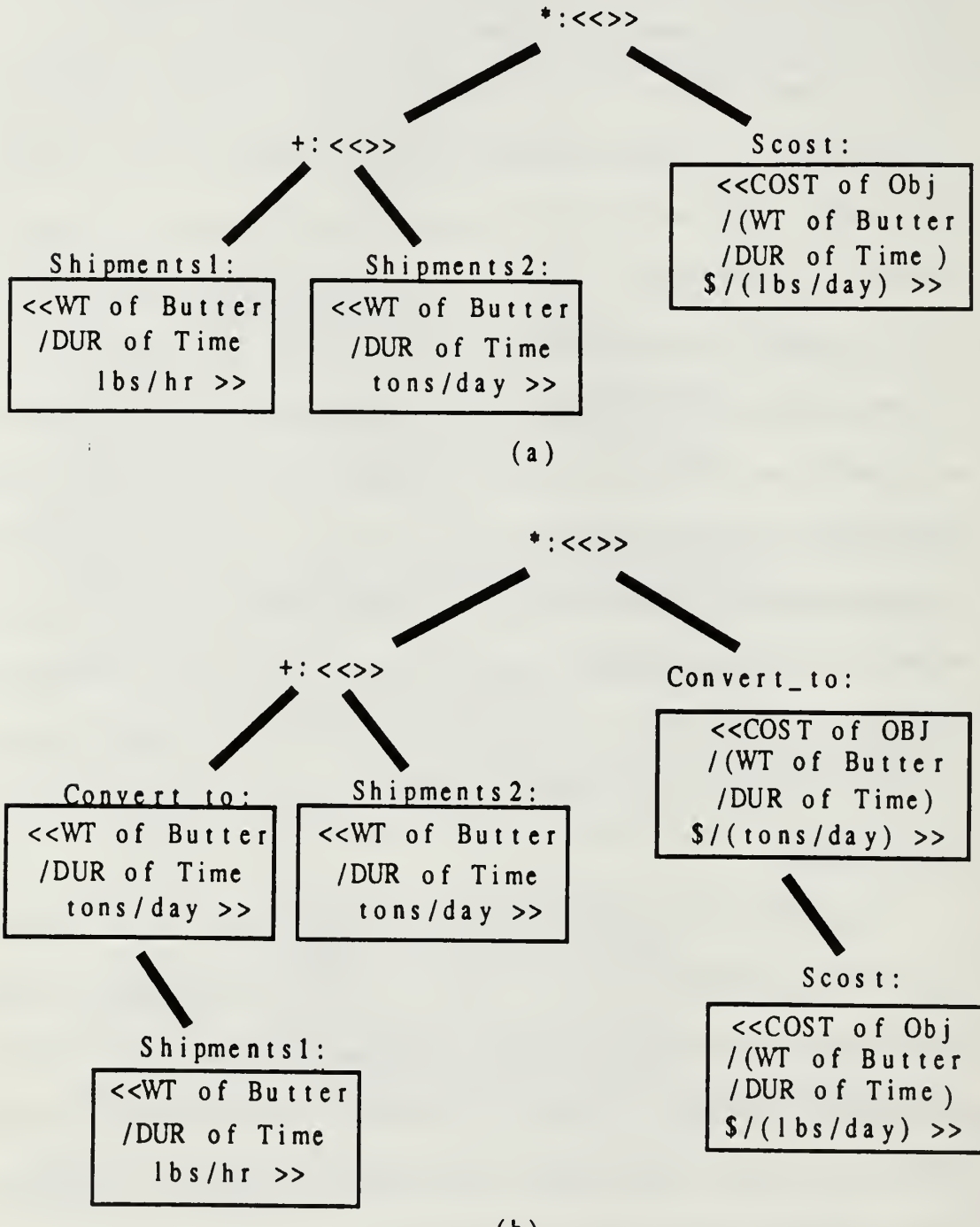


Figure 4-5. PostFix Expression Before and After Conversions

V. CONCLUSION

The results of this research can be viewed in two ways:

(1) As an actual, useable design for a type evaluator that can be included in any application that requires this function or
(2) A demonstration of the benefits in taking the program family approach in any design task.

By defining a domain and abstracting out the functions that are common to problems from the domain, we have been able to construct a core of routines that will allow the generation of a program family. While only three specific members of the family were implemented, each example member of the family used all of those routines, and the routines identified as non-changing were in fact used without any modifications. In addition, those routines that required modifications for the specific example were built only on the constraint that they adhere to the requirements of name and interface definitions. The internal functions of these routines had no bearing on the success of using the routines in conjuction with the core routines. The actual implementation of the examples was done only under the assumption that each example would involve declarations and expression evaluation as well as use a symbol table of pre-defined structure. By designing the elements of the core program to support the functions of declarations and evaluation in the abstract, and defining functions closer to each application instance at the name and interface level, we allow the user of the core program elements to focus on the details of choosing a language and the data structures

that best suit the typing requirements of the application. This can become a step towards building executable modeling languages, or it can be used in a language design environment. This can also be an example of the mechanics of involved in generation of a program family, and the benefits of such an approach.

One of the goals of software engineering is to make design of application programs less costly in terms of time spent in development as well as money; the program family approach should be considered seriously at the outset of any design task. Whether the design effort is expected to result in follow-on family members or not, using this approach can help in the top-down definition of the design task. Modules and interfaces are constructed that are closer to the abstract functions of the application, allowing easier design modifications of data structures during program development. Another benefit is that the overview of a design is more easily communicated to the design team if the initial design includes identified non-changing elements at a functional level. Truly reusable software can become more of a reality if the program family approach becomes the basis for design. Within the Department of Defense, the ADA programming language has been promoted because ADA modules can be compiled separately, stored in a library, and used by any programming team. We view the program family approach as a significant way to generate these reuseable modules.

LIST OF REFERENCES

[1] Bradley, G. H. and Clemence, R. D. Jr., "A Type Calculus for Executable Modeling Languages ," Technical Report NPS52-87-029, Naval Postgraduate School, Monterey, California, July 1987.

[2] Parnas, D. L., "On the Design and Development of Program Families ," *IEEE Transactions on Software Engineering Vol. SE-2, No. 12*, (March 1986).

[3] Calingart, P., in *Assemblers, Compilers, and Program Translation* , (Computer Science Press, Inc., 1979).

[4] Aho, A. L. and Ullman, J. D., in *Principles of Compiler Design* , (Addison-Wesley Publishing Co. , July 1987).

[5] Bradley, G. H. and Clemence, R. D. Jr., "A Type Calculus for Executable Modeling Languages ," Technical Report NPS52-87-029, Naval Postgraduate School, Monterey, California, July 1987.

[6] Parnas, D. L., "Designing Software for Ease of Extension and Contraction ," *IEEE Transactions on Software Engineering* (March 1979).

[7] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM Volume 15, No. 12*, (December 1972).

[8] Gotz, A., in *Introduction to Interval Computations*, (Academic Press, 1983).

[9] Clemence, R. D. Jr., "A Type Calculus for Modeling Languages ," PhD Dissertation, Naval Postgraduate School, Monterey, California, (In Progress).

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
5.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
6.	Professor Gordon H. Bradley, Code 52BZ Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	4
7.	Professor Daniel L. Davis, Code 52DV Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	1

Thesis
N175
c.1

Nachtsheim
Design and implementation of a program family for type evaluation.

Thesis
N175
c.1

Nachtsheim
Design and implementation of a program family for type evaluation.

thesN175
Design and implementation of a program f
3 2768 000 77518 3
DUDLEY KNOX LIBRARY